If you go to virtually any gathering of makers, you'll find Arduino microcontrollers at the core of dozens of projects. But not so much in ham projects. W8TEE wants to change that … and to make it easy for you to join in the fun.

Putting Arduino to Work in Your Ham Shack

BY DR. JACK PURDUM*, W8TEE

Unless you've been in a coma for the past decade or so, you already know about microcontrollers (µcs) and how they are finding more and more use in ham radio applications.[1] The question is: Are you using µcs in your own ham radio projects? If the answer is no, the next question is: "Why not?" It can't be cost. The Arduino Nano used in this article was purchased for $3.20, including shipping. The reason probably isn't the electronics, either, as the Arduino family of µcs are the electronics equivalent of Legos. Nope ... the real hangup that I hear most often is: "I don't know how to program." Really? Is that all that holding you back from a boatload of enjoyment?

Think back to when you got your ham license. I took my General Class license exam 60 years ago. Back then, I had to pass the 13-wpm code test. When you first thought about getting your ticket, did you know Morse code? If your exam was more recent, did you know what a Colpitts oscillator was or how to fix a ground loop problem? I sure didn't. But I learned. It's the same thing with µcs. In fact, there is enough open source software for the Arduino that you can use the µcs without having to learn at all!

The project presented here, *Photo A,* is a takeoff of one of the projects I built elsewhere.[2] It's a PS2 computer keyboard that can be used as a Morse code keyer. However, this version also includes a small 16 x 2 LCD. It's pretty well established that people learn Morse code faster when they listen for code patterns rather than individual letters. Also, many people learn more rapidly when they can see the letters as the code is being sent. This keyer lets you type in letters and have them instantly translated into Morse and sent to the rig, or you can buffer up to 127 keystrokes and play them after you're done typing. I find this useful during contacts when I listen to a long session but want to make sure I include some comment in my reply. I buffer my comments and then send it all when it's my turn. The keyer includes an optoisolator chip so you can easily interface it to your favorite rig. Therefore, you can use the keyer as a "true" keyer or as a code practice oscillator. An added benefit is that the keyer has a "perfect" fist.

Photo A. The Arduino-controlled keyboard/keyer project.

However, another goal of this article is to show just how easy it is to put together a few parts around an Arduino and come out with something that's pretty useful. The project was also selected because it attaches to external output devices that are controlled by the Arduino (e.g., a buzzer and a keyed circuit in a rig), reads data from an external input device (an inexpensive keyboard), plus uses a standard I/O interface (i.e., I2C) to control the LCD device. In other words, there's a lot of "stuff" going on, yet you may be surprised how easy it is to do all that.

Photo B shows the keyer before placing the LCD in the top half of the enclosure. The small prototype board holds an Arduino Nano, which is about 1.75 x 0.675 inches. The prototype board also contains a small piezo-electric buzzer (which serves as a sidetone), a 4N26 optoisolator, a few resistors, a PS2 connector, and connector pins for the LCD. With some wise Internet shopping, I was able to build the entire keyer for

* < jjpurdum@yahoo.com >

under $10. My enclosure is overkill, but it's what I had on hand, plus I wanted room for a 9-volt battery in case I decide to use it in the field.

Arduino Sources

If you are just getting started with using an Arduino, there are some great starter kits out there. I would check out the following:

- <http://bit.ly/1CtMCar>
- <http://bit.ly/1IjxOMW>
- <http://ebay.to/1C3tdcl>
- <http://amzn.to/1NFiOMK>
- <http://bit.ly/1HhKKT6>

All of these starter kits are top-notch and contain a wide variety of components, displays, and sensors. Some lean a bit towards robotics while others are more generalized. Shop around and find one that best suits your specific needs. Photo C shows some of the various Arduino boards. For a size reference, the Duemilanove board in the center of the photo is about the size of a deck of cards. The Nano is the second board from the bottom right in the photo.

The Dayton or similar hamfests carry a ton of components that can be used with the Arduino. While shopping, keep in mind that the code presented here works with all of the Arduino boards. I only selected the Nano because of its small size and low price.

The Circuit

There's not much to the circuit. *Figure 1* is a schematic for the keyer. I originally thought I might try using the ATTiny85 microcontroller or the Digispark (bottom right in *Photo C*, and the circuit is based on that chip). However, there is not enough memory on the '85 for the feature set in this keyer. (The '85 only has 8K of program memory and about 2K of that is chewed up by the bootloader. Think of a bootloader as a small program that lets your PC talk to the Arduino board.) Digispark has since come out with the Digispark Pro, which is about the size of a postage stamp and could be used in this circuit. In fact, you can substitute just about any Arduino-compatible board (e.g., ATMega168, 328, 1280, or 2560) and it will work. Whatever version you buy, make sure it has a USB connector on board, as that makes programming them a snap. The Arduino Pro Mini is cheaper, but programming it is a little trickier because it lacks the USB connector.

Arduinos comes with three flavors of memory: 1) flash, 2) SRAM, and 3) EEPROM memory. Flash memory is where your program gets stored. It's like a thumb drive in that its memory is non-volatile. That is, if you remove power, the program remains intact. The Nano has 32K of flash memory, which is almost triple what our program requires.

SRAM is where program data end up living while the program runs and it is volatile memory. When power is removed, SRAM goes stupid and forgets everything. The Nano has 2Kb of SRAM, which again is plenty for our needs.

EEPROM, or Electrically Erasable Programmable Read-Only Memory, is also non-volatile and is often used for storing configuration and other data that is needed, but not changed that often. One reason for reserving EEPROM for such data is because EEPROM memory can only go through so many read/write cycles before it gets flaky. The Nano has 1K of EEPROM, which we don't use. We could place the Morse code array in EEPROM, but there's no need to do so.

The PS2 connector (near the bottom-left in *Photo B*) is a PCB-mounted version, mainly because of its cost. You can find them on eBay for less than $1 apiece, quantity 5, including shipping.

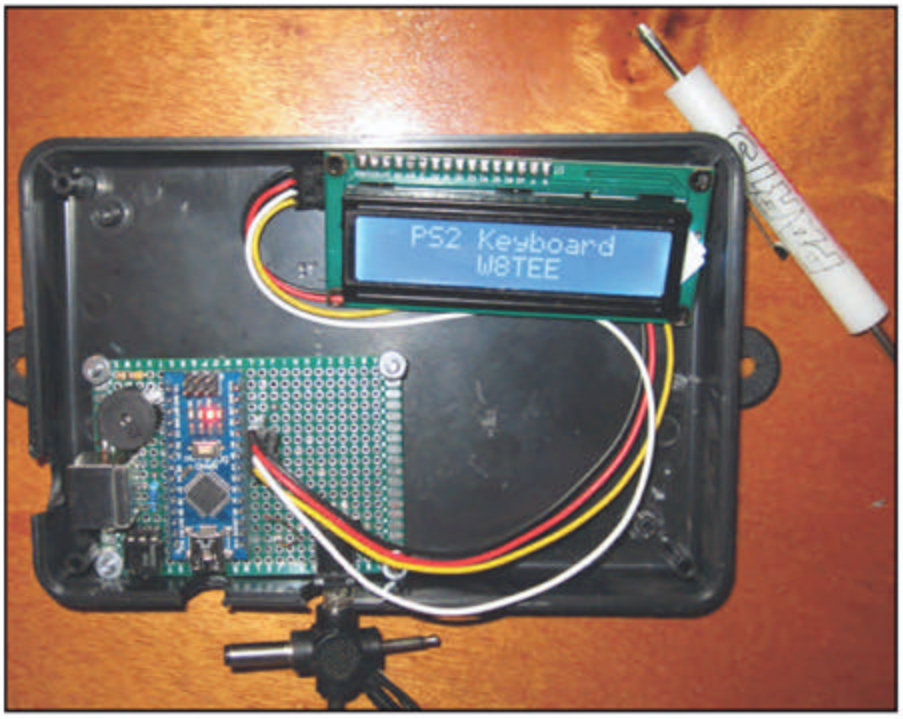

Photo B. The Arduino protoboard and LCD display powered by a 9-volt wall wart.

(The chassis-mount version of the connector is more than five times as much.) While PS2 keyboards have gone the way of the dodo, they are 1) easy to find, 2) easy to interface to, and 3) very inexpensive. I bought my keyboard at a church second-hand store for $2 ... works like a champ. No reason not to recycle them.

The I2C connector is simply four header pins and refers to the Inter-Integrated Circuit (pronounced I-squared-C) interface. This is a super simple two-wire interface with a well-documented library that is distributed with the Arduino software (more on that later). The LCD I used is from the Yourduino Starter Kit and comes with the I2C interface as part of the display. If you use a non-I2C LCD, you'll need to build the interface and add two pull-up resistors to the circuit (a good discussion can be found at <http://bit.ly/1xwrVLZ>). Alternatively, you could use a generic LCD and connect it with the standard interface (see <http://bit.ly/1IDKFSt>). The main reason for using the I2C interface is simplicity: It uses only two control pins on the Arduino, versus six pins for a standard interface. The standard interface and the code examples shipped with the Arduino software use digital I/O pins 12, 11, 5, 4, 3, and 2. You would have to move these common pin assignments to pins 12, 11, 9, 8, 7, and 6 because the PS2 connector uses pins 4 and 5 for the clock and data lines from the keyboard. Also, if you ever design a project that needs to use interrupt service routines (ISRs), most inexpensive Arduino boards only have two interrupt pins, numbers 2 and 3, so you may want to leave those free for the ISR routines. If you don't need any ISRs, feel free to use pins 2 and 3 as regular digital I/O pins.

By the way, it's usually best to leave pins 0 and 1 unused if at all possible, as these lines are the RX/TX data communications lines used for USB serial communications between the Arduino and the PC during programming. However, those pins may be used as regular digital pins after all USB communications are complete.

I used a 4N26 optoisolator to isolate

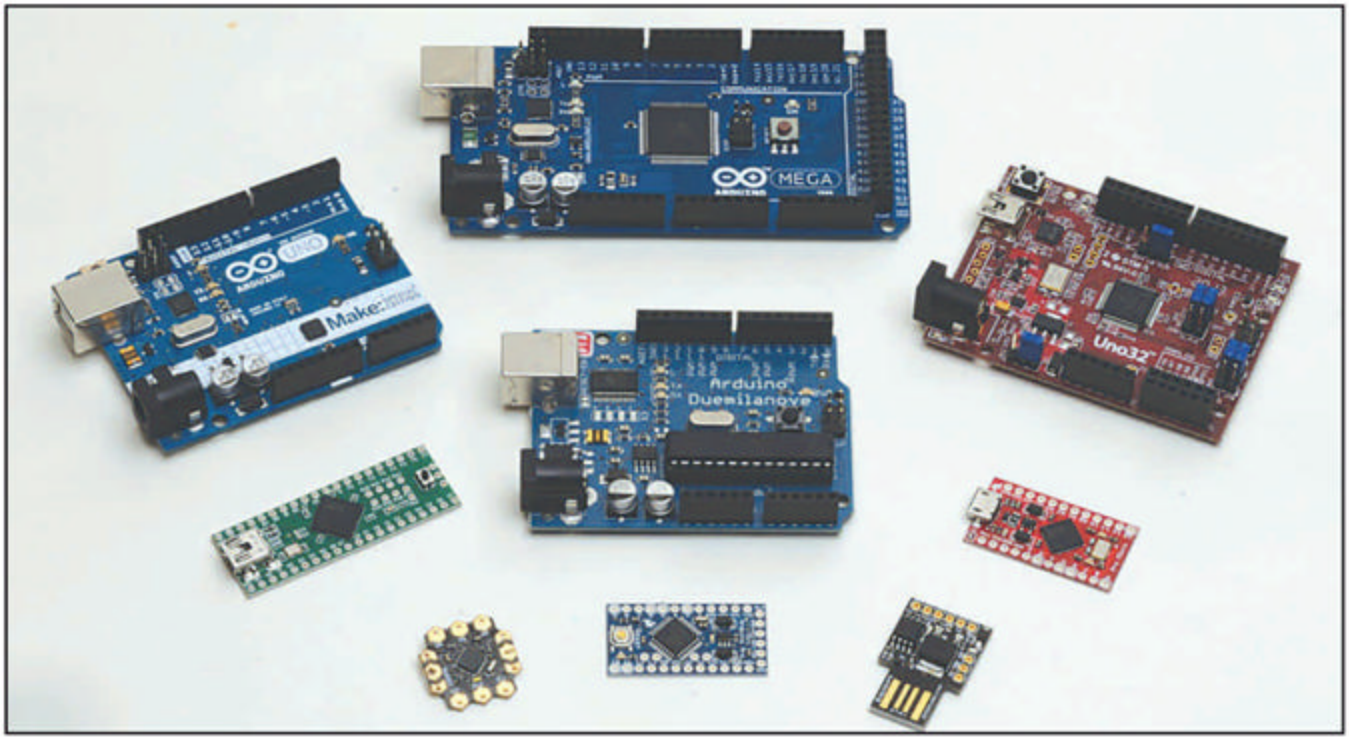

Photo C. Some Arduino-compatible boards.

bhi

Don't put up with noise and interference..
..Get a bhi DSP
Noise canceling product!

HEAR IT Speaker
- "Quick Adjust" DSP control
- 8 DSP filter levels 9 to 35dB
- 2.7W Amplified DSP speaker
- 3.5mm mono headphone socket
- On/off audio bypass switch
- 12 to 24V DC (500mA)

DSPKR

DSPKR -10 watt DSP speaker- 7 filter levels 9 to 24dB - Sleep mode - Filter store function - Volume control - Input overload LED - 3.5mm Mono headphone socket - 10 to 16V DC (2A) - Supplied with integral 3.5mm mono audio plug lead, user manual & fused DC power lead.

Come and see us in Dayton for a live demo at the GAP Antenna Booth!

DESKTOP

10 watt DESKTOP DSP speaker - 4" bass driver and 1" tweeter - Digital rotary volume and filter level controls - Separate stereo line-in and speaker level inputs - Headphone socket - Audio overload feature - Sleep mode - Noise reduction 9 to 35dB - 12V DC to 18V DC power (2.5A peak) - Weight 3.6lb, dims 8"(h) x 6"(d) x 6.3"(w) - Supplied with fused DC power lead, 3.5mm mono audio plug lead & user manual.

HEAR IT In-Line

HEAR IT In-Line - Amplified DSP In line module - Use with a speaker or phones - 8 filter levels 9 to 35dB - Separate input level and volume controls - 3 Watts audio - Supplied with 3.5mm mono audio plug lead, user manual & fused DC power lead

bhi also make a range of internal install modules to revive your radio or receiver

bhi Available from:
W4RT Electronics
fax: 256 880 3866
www.w4rt.com
info@w4rt.com

Products designed in the UK by bhi Ltd
E & O.E.
www.bhi-ltd.com

Available as "HEAR IT" from:
GAP Antenna Products Inc.
99 N.Willow St. Fellsmere, FL 32948
Tel: (772) 571 9922 Fax: (772)5719988
www.gapantenna.com
GAP

the Arduino from the rig's keying circuitry, but any 5-volt optoisolator should also work. There's nothing special about the piezo buzzer, either. The resistors are 1/8-watt and any value between 200 and 1K ohms will work for both resistors. For the keyed output, I just used a small phone jack which matches up with my QRP rig (*Photo B* was taken before I decided on how I wanted to handle the keyed line). You could put a switch on the battery line, but I usually just disconnect the battery when the keyer is not in use or I'm using a wall wart. The idle current drain is very small. While the Nano (as well as most other Arduino boards) has a built-in voltage regulator, there's plenty of room in the case for a 7805 and a few caps to provide regulation for any external power sources.

While we're here, keep in mind that the Arduino pins can sink a maximum of 40 mA each and no more than 400 mA total through the USB connector. To be on the safe side, I'd suggest a max of 20 mA on each pin and pay attention to the total current drain of the system. While an external power source may be able to supply more current than the USB port, always pay attention to the pin max. Once you let the smoke out of an Arduino, it's virtually impossible to put it back in.

The Software

The Arduino has its own programming environment called an Integrated Development Environment, or IDE. All of the tools you need to write a program (e.g., text editor, compiler, assembler, linker) are integrated into the Arduino IDE and operate more-or-less invisibly to the user. Best of all, it's free! Just go to the download website (<http://arduino.cc/en/Main/Software#toc3>) and select which version to download (i.e., Windows, Mac, or Linux). At press time, the current distribution version is 1.6.2, but new versions are released frequently. When I wrote this article, the version in use was 1.5.8 Beta.

Installing a Library

The program uses a special library written specifically for the PS2 keyboard. You can download it free from <http://bit.ly/1G49YqT>. This site also provides details on using the library. So, where should you install the new library so the IDE can access it?

The instructions that follow are for Windows, so you will have to adjust accordingly if you are using a Mac or Linux. I usually download a library ZIP file into a temporary directory named C://Temp (Pretty clever name, *n'est pas*?). When you download the PS2 library, the Temp directory will contain a file named *PS2Keyboard_2.3-Ctrl.zip*.

Load Windows Explorer and double-click the Zip file and Windows creates an "Extract all files" option near the top of Windows Explorer. Click that option to extract the files from the Zip file. After you extract the files, there is a new subdirectory named *PS2Keyboard_2.3-Ctrl*. Inside that directory is another new subdirectory named *PS2Keyboard*. It's the content of that directory that we need. Copy the PS2Keyboard directory to the *libraries* subdirectory of the IDE.

For my setup, I would copy the PS2Keyboard directory to *C://Arduino1.5.8/libraries* directory. You must restart the IDE for the new library to be recognized by the IDE. You can now use the PS2Keboard library files in your own programs by simply including its header file in your source code file using the *#include <PS2Keyboard.h>* directive in your code. (See the source code file.)

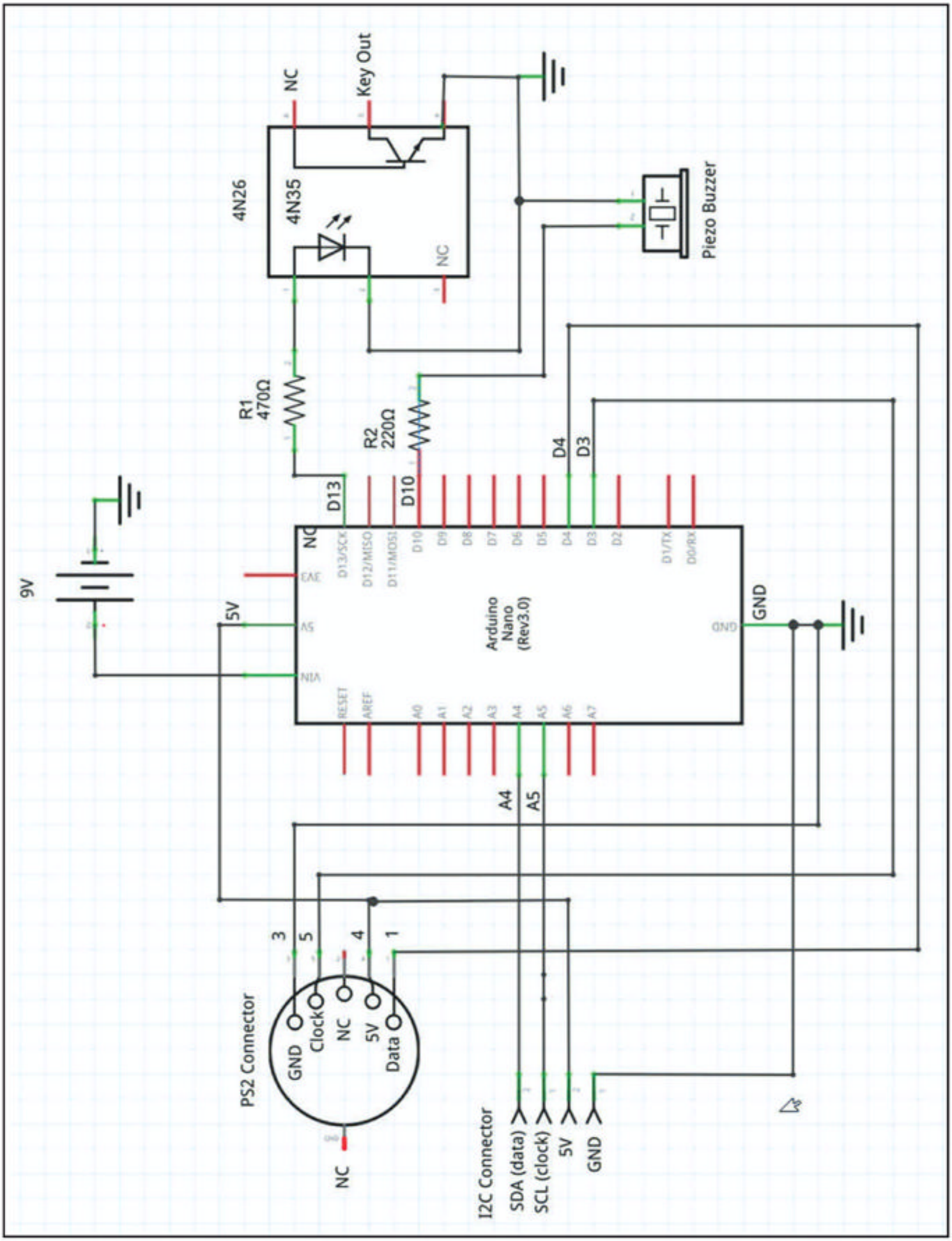

Figure 1. Schematic of PS2 keyer.

I usually create a directory for the IDE that reflects its current release number. For example, my IDE is installed on my C drive in a directory named Arduino1.5.8. Therefore, the path to the executable for the IDE on my system is C://Arduino1.5.8. You can place your IDE wherever you wish, but I find this method makes it easy to locate the directory when I need to access the IDE.

The IDE is distributed with a large selection of example programs. (Try the menu sequence: **File _ Examples _ Basics _ Blink** for a common example.) The examples are a great way to learn about programming the Arduino. If you've programmed before, most of this will be old hat to you. If you've never programmed at all, there are plenty of online sources from which to learn. I Googled "C programming tutorial" and got over 29 *million* hits. If you prefer a hands-on book, I'd suggest my book, *Beginning C for Arduino*.[3] It assumes absolutely no programming experience and has received good reviews.

When you run the IDE for the first time, your screen will look similar to *Figure 2*. As you can see, there are two default functions presented on startup: *setup()* and *loop()*. Every Arduino program *must* have these two functions. The *setup()* function contains the source code used to establish the operating environment in which the program runs. For example, if you want to print stuff on your PC from the Arduino as it runs, you would establish a serial link between the two in *setup()*. Other common activities in *setup()* include initializing arrays, timers, sensors, opening database or printer connections, and so forth. The important thing to keep in mind is that the code in *setup()* is only executed once, when the system is first powered up.

The *loop()* function contains the source code that you want to process repeatedly, in our case, reading the keyboard and processing the keystrokes. Most of the action associated with our program (and most others) takes place within the *loop()* function. Let's take a brief look at these two functions.

The *setup()* function source code for the PS2 keyer is presented in *Listing 1*. The first two program statements are used to set two of the Arduino pins to the output mode. Pin 13 on the Arduino is defined earlier in the program as LEDPIN. The Arduino family of boards has a small LED tied to pin 13 and can be used as an output indicator of your choice. Because this LED is integral to the Arduino board itself, it does not appear in the schematic in Figure 1. For our application, the LED pulses in sync with the code being sent, thus providing visual feedback on the keyer during testing. Pin 10 is defined as the TONEPIN, and is used to pulse the buzzer.

The next two statements initialize the keyboard (*kbd*) and display (*lcd*) objects. All of the code for these objects is packaged together in what is called a *library*. Most Arduino libraries are dedicated to the processing associated with some task or device. For these two libraries, those tasks are controlling the keyboard and LCD devices. There are hundreds of open source libraries written to control sensors, motors, servos, displays, and almost every other type of device you can think of. Obviously, every library means there's a body of existing code available to you that has already been written, tested, and debugged. Arduino projects go together quickly because you can stand on the shoulders of those who went before you. Plus, since you have access to the source code for these libraries, they are also a great learning tool. *See sidebar for tips on "Installing a Library."*

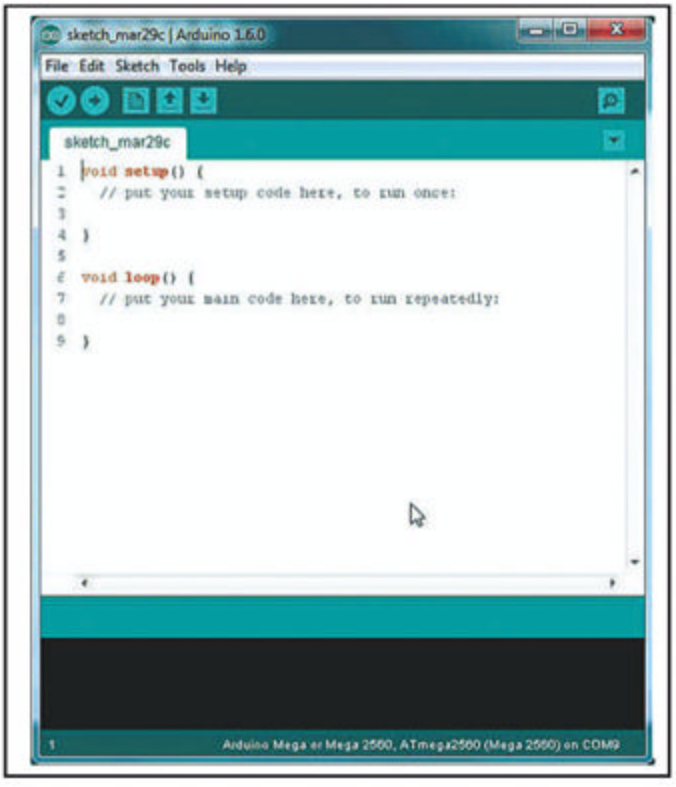
sketch_mar29c | Arduino 1.6.0
File Edit Sketch Tools Help
sketch_mar29c
void setup() {
// put your setup code here, to run once:
}
void loop() {
// put your main code here, to run repeatedly:
}
Arduino Mega or Mega 2560, ATmega2560 (Mega 2560) on COM9

Figure 2. Opening screen for Arduino IDE.

Listing 1. The setup() function

```
void setup()
{
  pinMode(LEDPIN, OUTPUT);
  pinMode(TONEPIN, OUTPUT);

  kbd.begin(PS2DATAPIN, PS2CLOCKPIN);  // initialize keyboard object with data & clock lines
  lcd.begin(LCDCOLSIZE, LCDROWSIZE);   // initialize lcd to 16x2 lines, turn on backlight

  lcd.setCursor(2,0);              // Splash screen
  lcd.print("PS2 Keyboard");
  lcd.setCursor(6,1);
  lcd.print("W8TEE");
  delay(2000);                          // Wait two seconds...
  lcd.clear();
  lcdColPosition = 0;
  bufferActive = 0;                     // Assume no buffering
}
```

The next four statements display a splash screen on the LCD object. After a 2-second delay, the LCD is cleared and some working variables are initialized. That's it! We have now set our program environment for this project.

As you might guess, the real work is done in the *loop()* function, which does everything else in the program. The *loop()* source code is presented in *Listing 2*.

Wait! There's literally nothing in the *loop()* listing! Actually, the *ps2poll()* function calls a host of other functions that end up doing the real work for the program. The complete code listing is too long to present here, but you can download it from <http://bit.ly/1CktvYB>. The code is well-commented so you should have little problem following it, even if you're not a programmer.

To use the program source code with your Arduino, first connect a USB cable from your PC to the USB connector on your Arduino board. (Most Arduino vendors supply the correct type of USB cable when you purchase the board.) Now use the **Tools _ Boards** menu sequence and select the Arduino board type you are using from the list of compatible boards. For me, it was the Nano board. Now use the **Tools _ Port** menu sequence to select a COM port for communications between the Arduino board and your PC using the USB cable connection. (This port uses the RX/TX pins for the communications I mentioned earlier.) Finally, load the source code file into the IDE **(File _ Open**), press Ctrl-U and the IDE compiles and uploads the code into your Arduino board. The program automatically begins execution when it finishes uploading the code. The USB cable supplies the voltage necessary to run the system. When you're happy with things, you can connect the battery or wall wart power source and disconnect the USB cable. The program code is stored in flash memory and will restart when power is applied again.

Features

Okay, you have everything up and running, so how do you use the keyer? Well, let's first give you a list of "commands" the program understands. These commands are presented in *Table 1*. Note that the commands presented in Table 1 are just those that I have implemented. You can add to that list just about anything that makes sense to you.

Conclusion

I hope you'll give the keyer project a try. If you want to enhance the keyer, it's simple to do. For example, if you have a message that is required for a contest, you could alter the % key message or even add an additional message. You would just have to find an unused key character (e.g., the $), go to the section of code where the special keys are processed, and add another case statement to cover your new key. There's enough spare SRAM that you should be able to add several new message (or other function) keys.

A slightly more difficult modification would be to sense another command key, perhaps an asterisk (*), and upon reading that key, the keyer starts generating a series of random letters and numbers until the next asterisk is read. (Hint: check the *randomSeed()* and *random()* library functions.) That's one of the great things about using a microcontroller: It's so easy to extend and modify what it can do. I should warn you, however, that once you start programming a microcontroller, there's no going back ... it's just too much fun.

Listing 2. The loop() function code.

```
void loop()
{
  ps2poll();                    // Loop looking for a keystroke

  if (bufferHead != bufferTail) {    // If there's a keystroke present in the buffer...
    send(BufferPopCharacter());      // ...send it along.
  }
}
```

Keystroke	Action
~	The tilde key toggles the buzzer. The default state is OFF. Press the tilde key and the buzzer changes state to ON. Press again and it reverts to OFF.
(	Start recording keystrokes. The keystrokes are written to a 127-character buffer. In the record mode, the buzzer is in the OFF state so as not to interfere with any code you may be listening to. If you attempt to write more than 127 characters, the buzzer sounds as an alarm to tell you keystrokes are no longer being recorded.
)	End the recording of keystrokes. The instant this key is read, the keyer starts sending the contents of the keystroke buffer. It continues until the buffer is empty.
#	Initiate/end a sending speed change. When this keystroke is read, the code prepares itself for either an increase or decrease in send ing speed. See next two commands. When the speed is set to the desired speed, a second # keystroke ends the speed change session.
>	Increase sending speed. Each key press increases the sending speed by one wpm.
<	Decrease sending speed. Each key press decreases the sending speed by one wpm.
%	This send a pre-recorded message. The software is shipped with a CQ message using my call. You may want to change that! The message is limited to 125 characters.
ESC	This clears the sending buffer. Useful when you have been placing a message in the send buffer, but decide you want to start over. It clears the buffer and resets the affected variables.

Table 1. PS@ Keyer Commands

Notes:

1. There are several books dedicated to Arduino projects for ham radio. Obviously, I like the book Dennis Kidder, W6DQ, and I wrote titled *Arduino Projects for Amateur Radio* (McGraw-Hill). An alternative is Glen Popiel's (KW5GP) book: *Arduino for Ham Radio* (ARRL).
2. Purdum and Kidder, Arduino Projects for Amateur Radio, pp. 173-198.
3. Purdum, *Beginning C for Arduino*, Apress Publishing, 2012. I'm obviously biased, but I do think it's a good learning experience.